HW-9

Question 1.

Explain each of the following techniques: *open addressing*, *separate chainning*, *linear probing*, *quadratic probing*, and *double hashing*.

Question 2.

Show the hash table of size 11 after inserting entries with keys 34, 29, 53, 44, 120, 39, 45, and 40, using each of the open addressing techniques.

Question 3.

(Implement hashCode for String) Write a method that returns a hash code for a string s using the formula:

$$\text{hashCode}(s) = s_0 \times b^{n-1} + s_1 \times b^{n-2} + \ldots + s_{n-1}$$

where $b = 31$. The function has the following header:

```
public static int hashCodeForString(String s)
```

Question 4.

Consider the class `MyHashMap` given below. The method `hash` can be defined as follows:

```
private int hash(int hashCode){
    return hashCode%capacity;
}
```

Choose a benchmark and compare the two implementations of `hash`. Analyze the result.

```
import java.util.LinkedList;

public class MyHashMap<K, V> implements MyMap<K, V> {
  // Define the default hash table size. Must be a power of 2
  private static int DEFAULT_INITIAL_CAPACITY = 4;

  // Define the maximum hash table size. 1 << 30 is same as 2^30
  private static int MAXIMUM_CAPACITY = 1 << 30;

  // Current hash table capacity. Capacity is a power of 2
  private int capacity;

  // Define default load factor
  private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;
```

```java
// Specify a load factor used in the hash table
private float loadFactorThreshold;

// The number of entries in the map
private int size = 0;

// Hash table is an array with each cell that is a linked list
LinkedList<MyMap.Entry<K,V>>[] table;

/** Construct a map with the default capacity and load factor */
public MyHashMap() {
  this(DEFAULT_INITIAL_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
}

/** Construct a map with the specified initial capacity and
 * default load factor */
public MyHashMap(int initialCapacity) {
  this(initialCapacity, DEFAULT_MAX_LOAD_FACTOR);
}

/** Construct a map with the specified initial capacity
 * and load factor */
public MyHashMap(int initialCapacity, float loadFactorThreshold) {
  if (initialCapacity > MAXIMUM_CAPACITY)
    this.capacity = MAXIMUM_CAPACITY;
  else
    this.capacity = trimToPowerOf2(initialCapacity);

  this.loadFactorThreshold = loadFactorThreshold;
  table = new LinkedList[capacity];
}

@Override /** Remove all of the entries from this map */
public void clear() {
  size = 0;
  removeEntries();
}

@Override /** Return true if the specified key is in the map */
public boolean containsKey(K key) {
  if (get(key) != null)
    return true;
  else
    return false;
}

@Override /** Return true if this map contains the value */
public boolean containsValue(V value) {
```

```
  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
        if (entry.getValue().equals(value))
          return true;
    }
  }

  return false;
}

@Override /** Return a set of entries in the map */
public java.util.Set<MyMap.Entry<K,V>> entrySet() {
  java.util.Set<MyMap.Entry<K, V>> set =
    new java.util.HashSet<>();

  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
        set.add(entry);
    }
  }

  return set;
}

@Override /** Return the value that matches the specified key */
public V get(K key) {
  int bucketIndex = hash(key.hashCode());
  if (table[bucketIndex] != null) {
    LinkedList<Entry<K, V>> bucket = table[bucketIndex];
    for (Entry<K, V> entry: bucket)
      if (entry.getKey().equals(key))
        return entry.getValue();
  }

  return null;
}

@Override /** Return true if this map contains no entries */
public boolean isEmpty() {
  return size == 0;
}

@Override /** Return a set consisting of the keys in this map */
public java.util.Set<K> keySet() {
```

```java
  java.util.Set<K> set = new java.util.HashSet<>();

  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
        set.add(entry.getKey());
    }
  }

  return set;
}

@Override /** Add an entry (key, value) into the map */
public V put(K key, V value) {
  if (get(key) != null) { // The key is already in the map
    int bucketIndex = hash(key.hashCode());
    LinkedList<Entry<K, V>> bucket = table[bucketIndex];
    for (Entry<K, V> entry: bucket)
      if (entry.getKey().equals(key)) {
        V oldValue = entry.getValue();
        // Replace old value with new value
        entry.value = value;
        // Return the old value for the key
        return oldValue;
      }
  }

  // Check load factor
  if (size >= capacity * loadFactorThreshold) {
    if (capacity == MAXIMUM_CAPACITY)
      throw new RuntimeException("Exceeding maximum capacity");

    rehash();
  }

  int bucketIndex = hash(key.hashCode());

  // Create a linked list for the bucket if it is not created
  if (table[bucketIndex] == null) {
    table[bucketIndex] = new LinkedList<Entry<K, V>>();
  }

  // Add a new entry (key, value) to hashTable[index]
  table[bucketIndex].add(new MyMap.Entry<K, V>(key, value));

  size++; // Increase size
```

```java
    return value;
  }

  @Override /** Remove the entries for the specified key */
  public void remove(K key) {
    int bucketIndex = hash(key.hashCode());

    // Remove the first entry that matches the key from a bucket
    if (table[bucketIndex] != null) {
      LinkedList<Entry<K, V>> bucket = table[bucketIndex];
      for (Entry<K, V> entry: bucket)
        if (entry.getKey().equals(key)) {
          bucket.remove(entry);
          size--; // Decrease size
          break; // Remove just one entry that matches the key
        }
    }
  }

  @Override /** Return the number of entries in this map */
  public int size() {
    return size;
  }

  @Override /** Return a set consisting of the values in this map */
  public java.util.Set<V> values() {
    java.util.Set<V> set = new java.util.HashSet<>();

    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        LinkedList<Entry<K, V>> bucket = table[i];
        for (Entry<K, V> entry: bucket)
          set.add(entry.getValue());
      }
    }

    return set;
  }

  /** Hash function */
  private int hash(int hashCode) {
    return supplementalHash(hashCode) & (capacity - 1);
  }

  /** Ensure the hashing is evenly distributed */
  private static int supplementalHash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
```

```java
  }

  /** Return a power of 2 for initialCapacity */
  private int trimToPowerOf2(int initialCapacity) {
    int capacity = 1;
    while (capacity < initialCapacity) {
      capacity <<= 1;
    }

    return capacity;
  }

  /** Remove all entries from each bucket */
  private void removeEntries() {
    for (int i = 0; i < capacity; i++) {
      if (table[i] != null) {
        table[i].clear();
      }
    }
  }

  /** Rehash the map */
  private void rehash() {
    java.util.Set<Entry<K, V>> set = entrySet(); // Get entries
    capacity <<= 1; // Double capacity
    table = new LinkedList[capacity]; // Create a new hash table
    size = 0; // Reset size to 0

    for (Entry<K, V> entry: set) {
      put(entry.getKey(), entry.getValue()); // Store to new table
    }
  }

  @Override
  public String toString() {
    StringBuilder builder = new StringBuilder("[");

    for (int i = 0; i < capacity; i++) {
      if (table[i] != null && table[i].size() > 0)
        for (Entry<K, V> entry: table[i])
          builder.append(entry);
    }

    builder.append("]");
    return builder.toString();
  }
}
```